CS3025 Compiladores

Semana 10:
Analisis Semantico:
Verificacion de Tipos / Typechecking
17 Octubre 2023

Igor Siveroni

Analisis Semantico: Verificacion de tipos / Typechecking

La fase de Analisis Semantico conecta las definiciones de variables a sus usos, verifica que cada expresión tenga el tipo correcto y, en algunos casos, traduce la sintaxis abstracta a una representación amigable apta para la generación de código.

Tablas de Simbolos:

- Esta fase se caracteriza por el mantenimiento de tablas de símbolos (symbol tables), también llamadas ambientes (environments), que mapean identificadores (variables, funciones, etc) a sus tipos, posiciones en memoria o cualquier otra información útil generada durante las fases de un compilador.
- Cada vez que el compilador procesa declaraciones de variables, tipos o funciones, el compilador asocia identificadores a *significados* en las tablas de símbolos. Cuando se encuentra un **uso** de un identificador, el compilador busca la entrada del identificador en la tabla de símbolos.
- Cada variable en un programa tiene un ámbito o *scope* en el cual es visible.

En el caso de **verificación de tipos**, la tabla de símbolos asocia nombres de variables y funciones a sus tipos.

El lenguaje IMP con declaraciones de funciones: sintaxis

Hemos estado trabajado con el lenguaje IMP, un lenguaje con declaraciones de funciones globales, definido por la siguiente sintaxis:

```
Program ::= VarDecList FunDecList
VarDecList ::= (VarDec)*
FunDecList ::= (FunDec)+
FunDec ::= ”fun” RType id “(“[ParamDecList]“)” Body “endfun”
ParamDecList ::= Type id (“,” Type id)*
VarDec ::= “var” Type VarList “;”
Type ::= bool | int
Rtype :: bool | int | void
Body ::= VarDecList StmList
StmList ::= Stm (“;” Stm)*
```

Donde un programa es una lista de declaraciones de variables globales seguidas de declaraciones de funciones. La declaración de una función especifica el tipo de retorno, el nombre y tipos de los parámetros, y el cuerpo definido como un bloque (Body) de declaraciones y sentencias.

El lenguaje IMP: sentencias y expressiones

La gramática de sentencias es:

```
Stm ::=  AssignStm | PrintStm | IfStm | WhileStm | ReturnStm
AssignStm ::= id “=” Exp
PrintStm ::= “print” “(“ Exp “)”
IfStm ::= “if” CExp “then” Body [“else” Body] “endif”
WhileStm ::= “while” Exp “do” Body “endwhile”
ReturnStm ::=“return” “(“ [Exp] “)”
```

La gramática de expresiones, con cuestiones de asociatividad y orden de precedencia resueltas, es:

```
Exp ::=  NumberExp | BoolExp | IdExp | CondExp | FCallExp | ParenthExp | BinExp
NumberExp ::= num     BoolExp ::= “true” | “false”     IdExp ::= id
BinExp ::= Exp op Exp, op in {plus, minus, mul, div, lt, leq, eq }
CondExp ::= “ifexp” “(“ Exp “,” Exp “,” Exp “)”
ParenthExp ::= “(“ Exp “)”
FCallExp ::= id “(“ [ArgList] “)”
ArgList ::= Exp (“,” Exp)*
```

El Sistema de tipos (type system) de IMP

El sistema de tipos de un lenguaje esta definido por lo menos por:

- El conjunto de tipos validos en el lenguaje (tipos base y reglas para generar nuevos tipos)
- Las reglas que determinan si un programa tiene los tipos correctos (equivalencia de tipos, subtyping, inferencia de tipos) i.e. “if a program is correctly typed”.

IMP trabaja con tipos base y tipos de función. Los tipos base son int y bool. Los tipos de funcion están hechos por una lista de tipos base (parámetros) y el tipo de retorno base (que además puede ser void). Denotaremos a los tipos con la variable T.

T in Type ::= BaseType | FunType
BaseType ::= int | bool
FunType ::= $BaseType^n$ x Rtype, n >= 0
Tr in RType :: int | bool | void

Escribiremos por ejemplo, T = $(T_1,\ldots,T_n)$ -> T_r

IMP: Tabla de simbolos / Type Environment

Durante el proceso de verificación de tipos mantendremos siempre a la mano una tabla de símbolos o environment de tipos, la cual asociara las variables “vivas” (variables in scope o validas en el ámbito actual) a sus tipos respectivos. Se contaran también los nombres de funciones.

El dominio de environments validos esta definido por:

`env in TEnv ::= id -> Type` (la -> se usa para definir funciones)

Usaremos [] para denotar al environment vacío. Además, las operaciones de lookup y update están definidas por:

- `env(x)` : retorna el valor de x guardado en env (lookup).
- `env(x)` **es valido** si `x in Dom(env)`, es decir, si x esta en el dominio de env.
- `env[x -> T]` actualiza el environment con el nuevo mapping [x -> T].
  Entonces: `env[x -> T](x) = T`
  Puede usarse: $\texttt{env}[x_1 \texttt{ -> } T_1]\ldots[x_n \texttt{ -> } T_n]$

IMP: Type checking expressions

Para facilitar la especificación de las reglas de typechecking, expresaremos la sintaxis de IMP usando un tipo especial de sintaxis abstracta (abstract syntax). Empezamos con las expresiones:

```
e in Exp ::=
      | NumberExp(n), n in Int
      | BoolExp(b)  , b in { T, F }
      | IdExp(id)   , id in ID
      | CondExp(e0, e1, e2) |
      | BinExp(e1, e2, op) ,
                op in {plus, minus, mul, div, lt, leq, eq }
      | FCallExp(fname, args) , args in Expn, fname in ID
      | ParenthExp(e)
```

Cuando trabajemos con listas o sequencias, e.g. `args`, podemos usar `|args|= n` para extraer la longitud `n` de la secuencia, o escribir `args = e1,…,en`. Tambien podremos expresar lo siguiente: `forall e in args, P(e).`

IMP: Type checking expresiones

Para especificar el type checking de expresiones definimos el dominio de la función typecheck de la siguiente manera:

```
tcheck: Tenv x Exp -> Type
```

y escribimos `tcheck(env, e) = T` para afirmar que la expresión `e` esta tipeada correctamente (cumple las reglas de tipeo, *correctly typed*) bajo en environment `env`, y que, además, tiene el tipo `T`.
Nótese que también pudimos haber definido typecheck como un predicado:

```
tcheck: Tenv x Exp x Type -> Bool
```

Ahora podemos definir typecheck por casos basados en la sintaxis abstracta de Exp:

```
tcheck(env, NumberExp(n)) = int

tcheck(env, BoolExp(b)) = bool

tcheck(env, IdExp(id)) = env(x)
```

IMP: Type checking expresiones

```
tcheck(env, ParenthExp(e)) = tcheck(env, e)

tcheck(env, CondExp(e0, e1, e2)) = T ifi
      tcheck(env,e0) = bool && T = tcheck(env,e1) = tcheck(env,e2)

tcheck(BinExp(e1,e2,op)) = T  ifi
      tcheck(env,e1) = int && tcheck(env, e2) = int
      (op in {plus,minus,mul,div} => T = int) &&
      (op in {lt, leq, eq } => T = bool)
```

Notar que hay varias maneras de especificar las reglas para BinOp. Por ejemplo, especificando mas casos de la forma,

```
      tcheck(BinExp(e1,e2,op)) = int    ifi
o     tcheck(BinExp(e1,e2,plus)) = T    ifi …
```

Dejamos FCall para el final.

IMP: Type checking statements (sentencias)

Del mismo modo que lo hecho con expresiones, expresaremos la sintaxis de las sentencias usando sintaxis abstracta (abstract syntax):

```
s in Stm ::=
       | AssignStm(id,e)
       | PrintStm(e)
       | IfStm(e, bd1, bd2) , bd1, bd2 in Body
       | WhileStm(e, bd)    , bd in Body
       | ReturnExp(e)

b in Body    ::= Body(vdlist, slist) , vdlist in VardDec*
vd in VarDec ::= (T, id)
```

IMP: Type checking statements (reglas)

Para el caso de sentencias y bloques (Body), definimos el predicado tcheck de la siguiente manera:

```
tcheck: TEnv x Stm -> Bool      tcheck: TEnv x Body -> Bool
```

Y escribimos `tcheck(env, s)` si la sentencia `s` es correcta, en lo que se refiera a tipos, dado el environment `env`. Igual para `tcheck(env, bd)`.
Especificamos `tcheck` para sentencias por casos basado en la sintaxis de `Stm`:

```
tcheck(env, PrintStm(e)) ifi tcheck(env,e)

tcheck(env, AssignStm(id,e)) ifi tcheck(env,e) = env(id)

tcheck(env, IfStm(e, bd1, bd2)) ifi
      tcheck(env,e) = true && tcheck(env, bd1) && tcheck(env, bd2)
```

IMP: Type checking statements (reglas)

Y escribimos `tcheck(env, s)` si la sentencia `s` es correcta, en lo que se refiera a tipos, dado el environment `env`. Igual para `tcheck(env, bd)`.
Especificamos `tcheck` para sentencias por casos basado en la sintaxis de `Stm`:

- `tcheck(env, WhileStm(e, bd))` **ifi** `tcheck(env,e) = true &&`
  `tcheck(env,bd)`

- `tcheck(env, Body(nil, slist))` **ifi**
  `forall s in slist: typecheck(env1, s)`

- `tcheck(env, Body(vdlist, slist))` **ifi**
  `forall s in slist: typecheck(env1, s), donde`
  `env1 = process_decs(env,vdlist)`

  `process_decs(env,vdlist) = env[`id_1 `->` T_1`],…,[`id_n `->` T_n`]`
  `donde vdlist =(`T_1`,`id_1`),…,(`T_n`,`id_n`)`

IMP: Type checking Programs y Declaraciones

Del mismo modo que lo hecho con expresiones y sentencias, expresaremos la sintaxis de Porgrams IMP y declaraciones de funciones usando sintaxis abstracta (abstract syntax):

```
p in IMP ::= Program(vdlist, fdlist),
                  vdlist in VarDec*, fdlist in FunDec*
fd in FunDec ::= FunDec(fname, pdlist, Tr, bd),
                  pdlist in ParamDec*, T in Rtype, bd in Body
pd in ParamDec ::= (T,id)
```

Y extendemos la definicion de tcheck a programas y declaraciones de funciones.

```
tcheck: TEnv x Program -> Bool
tcheck: TEnv x FunDec -> Bool
```

IMP: Type checking Programs y Declaraciones

Continuamos con la especificaion de `tcheck`:

```
tcheck(env, Program(vdlist, fdlist)) ifi
      forall fd_i in fdlist,
         let fd_i = FunDec(fname_i, pdlist_i, Tr_i, bd_i)
             Tf_i = funtype(pdlist_i, Tr_i)

      env1 = process_decs(env, vdlist)
      env2 = env1[fname_1 -> Tf_1],…,[fname_n -> Tf_n]

      forall fd in fdlist: tcheck(env2, fd)
```

IMP: Type checking Programs y Declaraciones

Continuamos con la especificaion de `tcheck`:

```
tcheck(env, FunDec(fname, pdlist, Tr, bd)) ifi
      Let pdlist = (T1,idn),…,(Tn,idn)
            env1 = env[id1 -> T1],…,[idn -> Tm][return -> Tr]
            tcheck(env1, bd)

tcheck(env, ReturnStm(e)) ifi
      Tr = env(return)
      Te = void if (e=nil) else tcheck(env,e)
      Tr = Te

tcheck(env, FCallExp(fname, args)) = T ifi
      Tf = env(fname) && Tf = (Tp1,…,Tpn) -> Tr
      forall ei in args: tcheck(env,ei) = Ti
      n = |args| && forall i in 1:n, Ti = Tpi
      Tr = T &&
```